Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

Theoretical Computer Science 302 (2003) 191–210

Theoretical
Computer Science

www.elsevier.com/locate/tcs

Optimizing stable in-place merging☆

Jingchao Chen

Department of Communication, Donghua University, 1882 Yan-an West Road, Shanghai, 200051, People's Republic of China

Received 8 April 2002; received in revised form 3 October 2002; accepted 4 October 2002
Communicated by J. Díaz

Abstract

In 2000, Geffert et al. (Theoret. Comput. Sci. 237 (2000) 159) presented an asymptotically efficient algorithm for stable merging in constant extra space. The algorithm requires at most $m_1(t+1)+m_2/2^t+o(m_1)$ comparisons ($t=\lfloor\log_2(m_2/m_1)\rfloor$) and $5m_2+12m_1+o(m_1)$ moves, where m_1 and m_2 are the sizes of two ordered sublists to be merged, and $m_1 \leqslant m_2$. This paper optimizes the algorithm. The optimized algorithm is simpler than their algorithm, and makes at most $m_1(t+1)+m_2/2^t+o(m_1+m_2)$ comparisons and $6m_2+7m_1+o(m_1+m_2)$ moves.

Keywords: In-place algorithms; Stable merging; Sorting

1. Introduction

Merging is a fundamental operation of mergesort. Its goal is to merge two adjacent sorted sequences of m_1 and m_2 elements into one sorted sequence of $n=m_1+m_2$ elements. Two important notions related to merging algorithms are *in-place* and *stable*. An algorithm can be regarded as *in-place* if it uses only a constant amount of extra space. Alternatively an algorithm can be regarded as *stable* if it keeps the initial relative order of elements with equal keys.

The reason why merging is studied is that a stable sorting algorithm which uses less time than order n^2 and at most $O(\log n)$ additional space had not been found, while Knuth was preparing his book about sorting algorithms [6]. To be able to devise an efficient stable in-place sorting algorithm, many researchers reduced this problem to searching for a stable in-place merging algorithm with linear time. In 1969, Kronrod

☆ This work was partially done while the author served in Bell Labs Research China, Lucent Technologies.

E-mail address: chen-jc@dhu.edu.cn (J. Chen).

0304-3975/03/$ - see front matter
PII: S0304-3975(02)00775-2

[7] developed the first unstable, and in-place, merging with linear time. Since then a lot of progress had been achieved on this problem. In 1977, Trabb Pardo [11] gave the first stable algorithm for in-place merging using the internal buffer technique introduced by Kronrod. Later this algorithm was somewhat simplified by Salowe and Steiger [9]. Irrespective of former or latter, the two algorithms are only theoretical breakthroughs since they are quite elusive structures and have rather large time-complexity constants of proportionality. Therefore, exploring simpler stable algorithms with lower complexities becomes a more recent research focus. In this aspect, the representative algorithms are Huang and Langston's algorithm [4], Symvonis's algorithm [10] and the algorithm of Geffert et al. [2]. The common feature of these algorithms is to construct a stable version with an unstable version. Huang and Langston's algorithm [4] is based on [3] and performs $1.5n + o(n)$ comparisons and $5.5n + o(n)$ exchanges, which is equal to $16.5n + o(n)$ moves since each element exchange needs three moves. The number of comparison required by the algorithm can be improved but still is bounded by $1.125n + o(n)$. Symvonis's algorithm [10] is based on Mannila and Ukkonen [8] and performs asymptotically the optimal number of comparisons but has no clear concept with respect to the constant factor in the number of moves required. To seek the lowest time-complexity, Geffert et al. developed two merging algorithms, one which is unstable and one which is stable. The unstable one requires at most $m_1(t+1) + m_2/2^t + o(m_1)$ comparisons ($t = \lfloor \log(m_2/m_1) \rfloor$) and $3(m_1 + m_2) + o(m_1)$ moves, where m_1 and m_2 are the sizes of the two input sequences. The stable one is obtained by making some modifications of the unstable one and performs the same number of comparison as the unstable one, but the number of moves is bounded by $5m_2 + 12m_1 + o(m_1)$.

Unlike the algorithms mentioned above, our algorithm has only a stable version without an unstable version. The core of our algorithm involves block rearranging, which divides the original problem into some subproblems, and local merging which solves the subproblems. In the case that the first sequence has more than $5n^{1/3}$ distinct keys, block rearranging exploits the simplified version of the unstable algorithm by Geffert et al., and local merging uses Huang and Langston's algorithm. Otherwise, block rearranging exploits selection sort, and local merging uses the algorithm of Geffert et al. The main difference between our algorithm and the existing algorithms is block size. The block size used by most algorithms is $\theta(n^{1/2})$. However, Geffert et al. [2] suggested using a larger block size, say $\theta(n^{2/3}/(\log n)^{1/3})$. Conversely, we think that a smaller block size, say $\theta(n^{1/3})$, can obtain a better performance. Our algorithm is the first to use the smaller block size. The number of comparisons required by our algorithm is bounded by $m_1(t+1) + m_2/2^t + o(n)$, where $t = \lfloor \log(m_2/m_1) \rfloor$, and the number of moves is bounded by $6m_2 + 7m_1 + o(n)$. Another significant improvement is that our algorithm is simpler than the algorithm of Geffert et al.

2. Basic concepts and techniques

2.1. Basic concepts and notation

In this paper, we assume that an element consists of a key and some data associated with that key. Let $K(x)$ denote the key of element x. x_i is smaller than or equal to x_j,

denoted by $x_i \leqslant x_j$ if and only if $K(x_i) \leqslant K(x_j)$. The smaller of two elements x_i and x_j will be denoted by $\mathbf{min}(x_i, x_j)$. A block U is a series of consecutive elements. $|U|$ will denote the number of elements in block U. The notation first(U) (last(U)) will refer to the first (last) element of U.

2.2. Internal buffering

Almost all in-place merging algorithms carry out block merging (also called local merging), using the internal buffering technique [2–4,7–11]. During block merging, the order of elements in some blocks is temporarily not important, such elements are called internal buffer elements. A group of locations occupied by these is called an internal buffer. Our algorithm will employ an internal buffer in block sorting and merging.

2.3. Swapping two blocks by the floating hole technique

The floating hole technique is due to Geffert et al. [2]. Now we outline this technique. Given two (not necessarily adjacent) blocks U and V of equal size and an empty location e (i.e. a hole) adjacent to the left of U, our task is to replace eU by Ve, and V by U, where the content of e can be ignored. This is easily achieved by shifting the ith V-element into the ith position of eU and then the ith U-element into the ith V-position, for $i = 1, \ldots, |U|$. The total number of moves required by this operation is precisely $2|U|$.

2.4. Block exchanging

Given two adjacent blocks U and V, the task of block exchanging is to place VU in the zone originally occupied by UV. Using Dudzinski and Dydek's technique [1], we can accomplish this task with $|UV| + \gcd(|U|, |V|)$ moves, where $\gcd(|U|, |V|)$ represents the greatest common divisor of $|U|$ and $|V|$. The basic principle may be briefly outlined as follows. Let $g = \gcd(|U|, |V|)$, we view UV as a series of $|UV|/g$ sub-blocks, each of size g. We use g cycles to perform this task. At the ith ($1 \leqslant i \leqslant g$) cycles, the ith element of each sub-block is moved to its final position. Since there are $|UV|/g$ sub-blocks, performing each cycle consumes $|UV|/g + 1$ moves. Thus, the overall number of movements required is $g(|UV|/g + 1) = |UV| + \gcd(|U|, |V|)$.

If the order of elements in one of two blocks is not important, by simply modifying this exchanging algorithm, we can reduce the number of moves. More precisely, it can be reduced to $|U| + |V| + 1$ if the order of elements in the larger-sized one of U and V is kept unchanged, and $2\mathbf{min}(|U|, |V|) + 1$ if the order of elements in the smaller-sized one of U and V is kept unchanged.

2.5. Block merging

In some special cases, we shall utilize the block merging technique described here. We assume that the input sequences to be merged are given in two consecutive blocks U and V, each with its keys in non-decreasing order. Here the procedure, which merges

U and V into one non-decreasing sequence, is called BLOCKMERGE. There are two different merging modes: forward and backward. A forward BLOCKMERGE may be described as follows. We locate the leftmost *insertion point* for the first element u_1 of U by scanning V from left to right. Thus, V is divided into two parts, namely, $V = V_1V'$, where $\text{last}(V_1) < u_1 \leqslant \text{first}(V')$ (If we use the rightmost *insertion point*, this formula should be $\text{last}(V_1) \leqslant u_1 < \text{first}(V')$). Then we exchange U and V_1 by the routine of Section 2.4, using $|UV_1| + \gcd(|U|, |V_1|)$ moves. The element u_1 and all elements to its left are now merged. This operation is repeated until U or V is exhausted.

Clearly, this procedure needs at most $|U| + |V|$ comparisons. But if we use binary search to determine the boundaries between the blocks to be exchanged, the number of comparisons can be bounded by $O(d \log |V|)$, where U contains d distinct elements. Since $\gcd(|U|, |V_1|) \leqslant |U|$, and each V-element is transported at most once, the total number of moves is at most $|V| + 2w|U|$, where w is the number of block exchanges.

A backward BLOCKMERGE is to merge the second block backward into the first. Its implementation details are similar to the forward one described above.

2.6. The binary merge of Hwang and Lin

The algorithm by Hwang and Lin [2,5] merges two sorted sequences of X and Y with at most $m_1(t+1) + \lfloor m_2/2^t \rfloor$ comparisons, which is close to the lower bound $m_1 t + \lfloor m_2/2^t \rfloor$, where $|X| = m_1, |Y| = m_2, m_1 \leqslant m_2$, and $t = \lfloor \log_2(m_2/m_1) \rfloor$ (Notice, if the optimization of $m_1(t+1) + \lfloor m_2/2^t \rfloor$ is not considered, t can be other non-negative integers. As will be seen in Section 4.2.2, t is fixed to be other constants). Let x_1 denote the first unmerged X-element, and y_i the ith unmerged Y-element for $i = 1, 2, \ldots, m$, where m is the number of the unmerged Y-elements. The algorithm may be briefly outlined as follows. First, we scan sequentially the $(0 \times 2^t + 1)$th, $(1 \times 2^t + 1)$th, $(2 \times 2^t + 1)$th, $\ldots$, $(i \times 2^t + 1)$th, and mth unmerged Y-element from left to right (i.e. use the space 2^t to sample the unmerged Y-elements) until the leftmost Y-element $y_j \geqslant x_1$, where $j = k2^t + 1$ (k is some positive integer), or $j = m$, is found. Second, we make a binary search over the $2^t - 1$ element to the left of y_j to locate the rightmost unmerged Y-element $y_h < x_1$. Finally, we move the unmerged Y-elements to the left of y_h to the output, followed by x_1. The process is repeated until X or Y is exhausted. After this is done, we directly append the rest of the remaining sequence onto the end of the output.

It is easy to verify that the number of comparisons required by the algorithm is equal to the above formula. Each X-element spends t comparisons to make the binary search, and one comparison to compare x_1 and y_j. In addition, in the sequential scan, we use one comparison for at least 2^t Y-elements transported to the output.

3. A variant of the unstable merge by Geffert et al.

In 2000, Geffert et al. [2] designed an unstable merging algorithm using at most $n + o(n)$ comparisons and $3n + o(n)$ moves, where n is the size sum of the two input sequences X and Y. In Section 4, we shall use the algorithm of Geffert et al. to sort

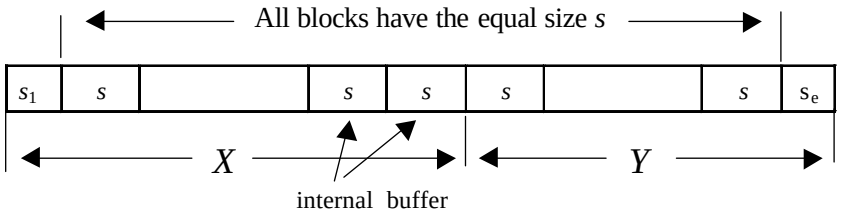


Fig. 1. Blocking X and Y.

blocks of size $O(n^{1/3})$. If we directly invoke this algorithm to do this, it is imperative to create the hole of size $O(n^{1/3})$. Obviously, this violates the restriction of constant workspace. That is, the *floating hole* technique in the algorithm of Geffert et al. is not suitable for our purpose. However, we can make use of the *floating hole* technique in a different way, as described in Section 3.2.

3.1. The unstable merge of Geffert et al. without the floating hole technique

As shown in Fig. 1, we divide the two adjacent input sequences X and Y into blocks of equal size s, except for the first X-block of size s_1 and the last Y-block of size s_e, where $s_1 = |X| \bmod s$ if $|X| \bmod s > 0$, and $s_1 = s$ otherwise, similarly $s_e = |Y| \bmod s$ if $|Y| \bmod s > 0$, and $s_e = s$ otherwise. In Section 4.2, it will be seen that the value of s equals $\tilde{n}^{1/2}$, where $\tilde{n}$ ($\approx n^{2/3} - 6$) is the total number of superelements to be merged (a superelement in Section 4.2, which is a block of size $n^{1/3}$, corresponds to an element here). The last two blocks of the X-sequence are used as two internal buffers. Initially, the second to last X-block is called the *current buffer* (denoted by B_1), the last X-block is called the *non-current buffer*, and the first X-block (Y-block) is called the *current X-block* (*Y-block*).

The blocks of X can be mixed up, (but not elements within these blocks excluding the buffer blocks), while the blocks of Y are not mixed, still forming a contiguous zone. The non-current buffer is optional, that is, whenever the current buffer is exhausted.

The basic idea of the unstable merge by Geffert et al. is to do with the merging of the current X- and Y-block and the selection of X-blocks alternately. In more detail, we merge the current X- and Y-block, using the current buffer. Once one of the following cases occurs: (1) the current X-block becomes empty; (2) the current Y-block becomes empty; (3) the current buffer becomes full, ..., etc., we re-determine a new current X-block, a new current Y-block or a new current buffer (here determining a new Y-block is simple because it is just the next Y-block), and resume the merging process until all elements are placed in their final position.

We denote the positions of the current X-, Y- and B_1-element by x_c, y_c and b_c, respectively. The start position of the *non-current buffer* is stored in a variable n_c. The block to be occupied by the final result is called the *output block*, and the current output position, located inside this block, is denoted by o_c.

At every moment the elements merged so far are to the left of o_c. The Y-elements yet to be merged are to the right of y_c in their original order. Between o_c and y_c we

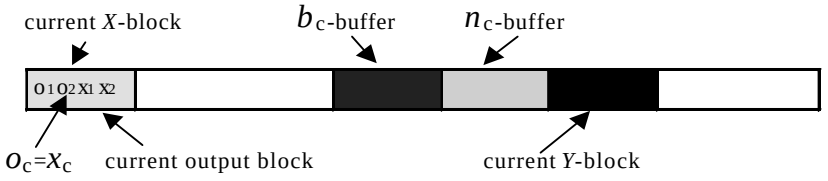


Fig. 2. A case $o_c = x_c$.

have the X-blocks yet to be merged, usually out of order. The buffer elements, which are the largest $2s$ X-elements, are located in any order in the current Y-block to the left of y_c, in the current X-block to the left of x_c, in the current buffer to the right of b_c, and in the non-current buffer if it exists. Sometimes the elements in the current buffer to the left of b_c and in the output block to the right of o_c form a logical X-block which is just been moved from the output block to the current buffer to make room for the sorted output.

Initially, the output block and current X-block are the same block. Assuming that X has indexes from 0 to $|X|-1$, we do the following initial setting: $o_c = x_c = 0$, $y_c = |X|$, $b_c = |X| - s - s_1$, and $n_c = |X| - s$. Clearly, this setting implies that $o_c \bmod s = b_c \bmod s$, which is for keeping o_c synchronized with b_c. For simplicity, x_c-element will refer to the element which x_c points to, and similarly for b_c, y_c, etc. The notation $o_c \leftarrow x_c \leftarrow b_c$ will denote that the x_c-element is moved to o_c-position and then the b_c-element is moved to x_c-position.

Below we construct the algorithm in state processes and function modules. Sections 3.1.1–3.1.3, 3.1.7 and 3.1.9 reflect state processes, while others (Section 3.1.4, 3.1.5, etc.) reflect function modules. A state process corresponds to a program segment. Usually, using a "go-to" instruction runs it. A function module corresponds to a routine with a "return" instruction. Using a "call" instruction runs it. The algorithm starts in Section 3.1.1.

3.1.1. Output position o_c equals the current X-position x_c

Fig. 2 shows a case when the current X-element x_c has the same position as the output position o_c. In this case, we find the leftmost element $x_k > y_c$-element (the current Y-element) over the current X-block from left to right. If no such x_k is found, i.e. the last element of the current X-block $\leqslant y_c$-element, we scan sequentially the remaining X-blocks to determine the new current X-block. Notice, the X-blocks from the right of the output block to the left of the current Y-block are out of order, so we cannot select the next X-block as the new current X-block. Instead, we choose the block with the smallest first key as the next to be processed. In the case of equal first keys, we use the last keys of the blocks as the secondary keys. When the last keys become equal, the leftmost one is selected. Once the new current X-block is found out, we have the first position of the new current X-block become the new x_c. If the new current X-block happens to be the next output block, we resume this process. Otherwise, the program control is switched to Section 3.1.3 or 3.1.7.

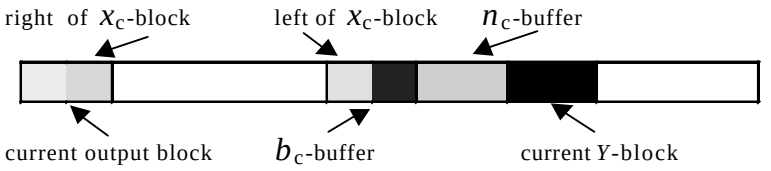


Fig. 3. Current X-block overlays current buffer.

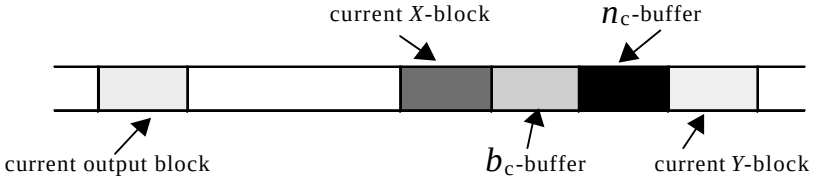


Fig. 4. Output block, current X-block and buffer block are all disjointed.

Once the x_k-element is found out, we make the following assignment operations: $o_c := x_k$, $b_c := \lfloor b_c/s \rfloor * s + (x_k \bmod s)$ so that $o_c \bmod s = b_c \bmod s$. Then we perform the exchange of the related elements by the following four element movements: $temp \leftarrow o_c \leftarrow y_c \leftarrow b_c \leftarrow temp$ where $temp$ is a temporary variable. x_c is set to be b_c. And then o_c, y_c, and b_c are incremented. The subsequent process is transferred to Section 3.1.2.

3.1.2. *Current X-block overlays current buffer*

As shown in Fig. 3, the current X-block spans across the output block and the current buffer, i.e. the right part of the current X-block is located in the output block, and the left part of the current X-block is located in the buffer block. The current buffer is broken into two pieces: the left of x_c-element and the right of b_c-element. We handle the case in the same way as described in Section 3.1.3. That is, we perform repeatedly the following four element movements:

$$temp \leftarrow o_c \leftarrow \mathbf{arg}\ \mathbf{min}_{z \in \{x_c, y_c\}} \{z\text{-element}\} \leftarrow b_c \leftarrow temp,$$

where $\mathbf{arg}\ \mathbf{min}_{z \in \{x_c, y_c\}} \{z\text{-element}\} = x_c$ when x_c-element $\leqslant y_c$-element and y_c otherwise. (In the rest of the paper, the notation is abbreviated as $\mathbf{arg}\,\mathbf{min}\{x_c, y_c\}$). Then the index variables o_c, b_c and x_c or y_c are incremented. The process is repeated until b_c reaches a block boundary. It is easy to see that b_c moves to the right "faster" than x_c, since the x_c-element is always on the left of b_c-element. When b_c reaches a block boundary, the subsequent process is transferred to Section 3.1.4.

3.1.3. *Output block, current X-block and buffer block are all disjointed*

Fig. 4 illustrates a case when the output block, the current X-block, the current Y-block and the buffer block are all disjointed. In this case we perform repeatedly the

following four element movements: $temp \leftarrow o_c \leftarrow \mathbf{arg\,min}\{x_c, y_c\} \leftarrow b_c \leftarrow temp$. Then all relevant index variables are incremented. The process is repeated until b_c, x_c or y_c reaches a block boundary. When b_c, x_c or $y_c$reaches a block boundary the subsequent process is transferred to Sections 3.1.4, 3.1.5 or 3.1.6.

3.1.4. Current buffer becomes full

When the current buffer becomes full, we have an entire non-current buffer block available. Suppose that no non-current buffer block is available. Then the current X- and Y-block both contains at most $s-1$ buffer elements. Thus we have at most $2s-2$ buffer elements in total. However, this is a contradiction since the total number of buffer elements should be exactly $2s$.

We let the non-current buffer become the new current buffer and accomplish this operation by assigning n_c (the beginning of the non-current block) to b_c.

3.1.5. Current Y-block becomes empty

Once the last element of the current Y-block is moved to the output block, we assign $y_c - s$ to n_c so that the current Y-block becomes the non-current buffer, then let the next element of y_c become the new y_c-element.

If two non-current buffers exist at the same time, the above operations could cause the buffer elements "lost". However, this case never happens. Assuming that we have two blocks containing $2s$ buffer element, namely, non-current buffer and the old Y-block. Then we have at least $2s+1$ buffer elements. Since the current buffer contains at least one buffer element. Nevertheless, we know that the total number of buffer elements is always equal to $2s$, which is a contradiction.

3.1.6. Current X-block becomes empty

Here the work to be done is similar to the case described in Section 3.1.5. That is, let the current X-block become the non-current buffer. However, since the remaining X-blocks get usually out of order, the way to determine the new current X-block is different from the Y-block case, but is the same as described in Section 3.1.1. That is, we scan sequentially the remaining X-blocks to determine the next block to be processed, using the first key and last key of each block as a key value. This scanning may be different depending on the case whether the current buffer is empty or not.

If the current buffer is not empty, the first block to be scanned should be a logical block that consists of the elements to the left of b_c in the current buffer and the elements to the right of o_c in the output block. So we have to use the first key in the current buffer and the last key in the output block as a starting key, then search the remaining X-blocks for a block with the smallest key. If the logical block is the next to be handled, the subsequent process is transferred to Section 3.1.2.

If the current buffer is empty, the first block to be scanned should be the output block. Then we search for the new current X-block in the usual fashion of jumping the current buffer. If the output block is the next block to be handled, the subsequent process is transferred to Section 3.1.1.

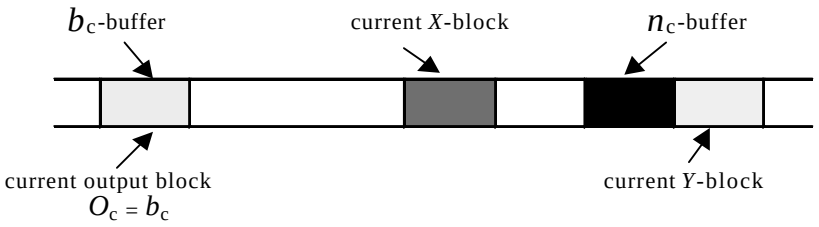


Fig. 5. Output block overlays current buffer.

3.1.7. Output block overlays current buffer

Because the algorithm always synchronizes o_c and b_c, i.e. $o_c \bmod s = b_c \bmod s$, when the output block overlays the current buffer, we have always $o_c = b_c$. Fig. 5 illustrates a case when $o_c = b_c$. In this case, we exchange the o_c-element and the smaller of x_c- and y_c-element, and then the relevant index variables are incremented. This process is repeated until o_c and b_c reach a block boundary. Then the program control will turn to Section 3.1.4.

3.1.8. Output block overlays non-current buffer block

It is easy to analyze that this may happen only if all block-control variables reach block boundaries at the same time. By a similar analysis to Section 3.1.7, b_c should point to the beginning of the current buffer when o_c points to the beginning of the non-current buffer block. Thus, we exchange the non-current buffer and the current buffer by swapping their pointers stored in n_c and b_c so that this becomes the case described in Section 3.1.7.

3.1.9. Output block overlays current X-block

If $o_c = x_c$, the program control turns immediately to Section 3.1.1. Otherwise, the current X-block contains at least one buffer element. By a similar analysis to Section 3.1.7, b_c should point to the beginning of the current buffer when o_c points to the beginning of the current X-block. Thus, we infer easily that there is no entire non-current buffer block available. Hence, we can let the current buffer become the non-current buffer, and then let the output block become the current buffer by setting b_c to o_c. The situation now is the same as in Section 3.1.7. However, there are two cases to be considered.

(A) When x_c reaches the block boundary, we start with the next to the current X-block and search sequentially the remaining X-blocks for the next X-block to be processed. Since the output block and current buffer are overlaid still, we continue the processing in the way of Section 3.1.7.

(B) When the output position o_c bumps into x_c, i.e. $o_c = x_c$, the subsequent process is transferred to Section 3.1.1.

3.1.10. Y-elements are exhausted

If the last Y-element has been placed in the final position, we append the remaining X-elements in the ascending order. Moving the X-elements is completely the same as

if we had y_c which points to $+\infty$, a dummy largest element. But actually the dummy y_c-element is not compared with the X-elements.

3.1.11. X-elements are exhausted

If the last X-element has been placed in the final position, the output result is in the form of ZBY', where Z denotes the merged output, B the two buffers, Y' a sequence of Y-elements not yet merged.

Now we merge B directly with Y' by selection sort. That is, we find the smallest element b_1 by scanning B from left to right, and then put $\mathbf{min}(b_1, y_1)$ into the final position by exchanging the o_c-element and $\mathbf{min}(b_1, y_1)$, where y_1 is the first Y'-element. This operation is repeatedly used to merge the remaining B- and Y'-elements until B or Y' is exhausted. Clearly, merging B with Y' takes at most $|Y'| + 2s(2s + 1)/2$ comparisons and $3(|Y'| + 2s)$ moves (Notice, $|B| = 2s$).

By an analysis analogous to [2], we can obtain that the unstable algorithm requires at most $|X| + |Y| + O(|X|^2/s^2) + O(s^2)$ comparisons and $4(|X| + |Y|)$ moves, since we take at most four moves per each element moved to the final position.

3.2. A new merging problem and its solution

Here we introduce a new merging problem, which is a variant of the original merging problem described in Section 3.1. We are now given an array of the form bXY, where X and Y are the sequences to be merged, and b denotes an extra buffer element at the beginning. The task is to replace bXY by $(X + Y)b$, where $X + Y$ denotes the sequence formed by merging X with Y. This problem will be seen in Section 4.2.1.

The feature of this problem gives us subtle insights into the floating hole implementation. To merge X with Y, the original floating-hole version of the algorithm of Geffert et al., which inspired the algorithm above, always creates a hole within the buffer each time we terminate such an operation: placing an X-element or Y-element in its final position. But here we always let the current output position become a hole. That is, we start the above algorithm by putting the element b adjacent to the left end of X aside to create a hole e. Of course, after this operation we let o_c point to the hole e. Then, we modify the relevant element movements in the above algorithm. For example, we replace $temp \leftarrow o_c \leftarrow y_c \leftarrow b_c \leftarrow temp$ by $o_c \leftarrow y_c \leftarrow b_c \leftarrow o_c + 1$, and the index variable o_c is incremented to point to a new hole. Similarly for the operations: $temp \leftarrow o_c \leftarrow \mathbf{arg\,min}\{x_c, y_c\} \leftarrow b_c \leftarrow temp$, and $o_c \leftrightarrow x_c$, etc. In Section 3.1.11, the operation such as exchanging the o_c-element with $\mathbf{min}(b_1, y_1)$ can be replaced by $o_c \leftarrow \mathbf{arg\,min}\{b_1, y_1\} \leftarrow o_c + 1$, and incrementing the index variable o_c.

Clearly, where there exist the element movements, we save one move per each element transported to the output. It is easy to see that using the floating hole technique to solve the new merging problem, the algorithm can save one fourth the number of moves, and keep the number of comparisons unchanged. Hence, by Section 3.1.11, the algorithm requires at most $|X| + |Y| + O(|X|^2/s^2) + O(s^2)$ comparisons and $3(|X| + |Y|)$ moves, where s is the size of blocks.

4. Stable in-place merging

The goal of this section is to merge two adjacent sorted sequences X ($|X| = m_1$) and Y ($|Y| = m_2$) stably in-place by the following four phases: Extraction of internal buffer, block sorting, local merging and buffer insertion. Depending on whether the number of distinct X-elements is above $\lambda = 5\lfloor n^{1/3} \rfloor$ ($n = m_1 + m_2$) or not, we adopt different block sorting and local merging strategies.

In Section 4.2, we will use two different kinds of "buffering". One consists of $2s$ buffer superelements, initially at the end of X, corresponding to the $2s$ buffer elements of Section 3.1. This will still be denoted by the notation "B". Another consists of some simple buffer elements, all distinct, used to keep the algorithm stable. This will be denoted by a new notation "C". More precisely, C consists of

C_1—the elements to be swapped with the first elements in each superblock, to remember the relative order of mixed superblocks in X.

C_2—the elements to be swapped with the first elements in each buffer superelement, to remember the relative order of mixed buffer superelements.

C_3—used for local merging.

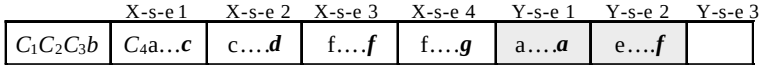
C_4—to align the first undersized X-block.

4.1. Extraction of buffer C

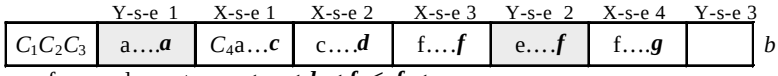
Here we attempt to extract no more than λ distinct elements from the X-sequence, then place them in the head of the X-sequence. The basic technique is the same as in [4], but we adopt a modified technique to exchange the buffer with the X-subsequence adjacent to the left of the buffer. During the extraction of the buffer, the X-sequence is usually of form Px_LEx_RCS, where P and S are the prefix and suffix of the X-sequence, C is the buffer obtained so far, x_R is the element adjacent to the left of C, x_L is the leftmost element that is equal to x_R, and E is the block of X-elements all equal to x_R, but excluding x_L and x_R. We find x_L by a binary search over the rest of X-sequence. If $x_L \equiv x_R$, that is, there is only one element equal to x_R, namely, the element x_R itself, we do nothing. Otherwise, we exchange Ex_R with C by the routine given in Section 2.4, in $|Ex_RC| + \gcd(|Ex_R|, |C|)$ moves. Viewing x_LC as the new C, Ex_RS as the new S, we resume the process until $|C| = \lambda$ or C reaches the head of the X-sequence. When $|C| = \lambda$, we move C to the left by invoking the routine of Section 2.4. Now the buffer C is sorted and located in the head of X.

Since each buffer element is selected by a binary search, extracting the buffer requires at most $|C|(\log_2 |X| + O(1)) \leqslant \lambda(\log_2 m_1 + O(1)) = O(n^{1/3} \log m_1)$ comparisons. Let k denote the total number of block exchanges excluding the last block exchange. Clearly $k < \lambda$. Since each X-element take part in at most one block exchange, and $|Ex_RC| + \gcd(|Ex_R|, |C|) \leqslant |Ex_R| + 2|C|$, the total number of moves required to extract the buffer is at most $e_\lambda + 2\lambda + \sum_{i=1}^{k}(e_i + 2i) \leqslant |X| + \lambda^2 + \lambda = m_1 + O(n^{2/3})$, where e_λ is the size of the block adjacent to the left of C at the movement of C to the left, and e_i denotes the size of block Ex_R at the ith block exchange.

The notations in the figure are defined below.
* s-e: superelement ; b: buffer element; C_1, C_2, C_3, C_4: buffer
* Except b, all small letters denote keys.
* Italics, i.e. the last letters of superelements, denote the keys of superelements.



(a) Input



Keys of superelements: $a < c < d < f \leq f < g$

(b) Output

Fig. 6. The superelement sorting of Section 4.2.1.

After the extraction of buffer, XY is transformed into $CX'Y$, where X' is the remaining X-elements that have not been selected as C-elements. Let β denote the number of distinct X-elements. Depending on whether $\lambda \leqslant \beta$ or not, we adopt different strategies to transform $CX'Y$ into the final result (the stable merging of X and Y). Section 4.2 describes how to transform $CX'Y$ into the final result when $\lambda \leqslant \beta$, i.e. the X-sequence has at least λ distinct keys, while Section 4.3 is the case when $\beta < \lambda$.

4.2. *Merging in the presence of at least λ distinct keys*

If the X-sequence has at least $\lambda = 5\lfloor n^{1/3} \rfloor$ distinct elements, we can create a buffer C, of size λ, that consists of only distinct elements, and transform X into CX'. The section focuses on how to use the C to transform $CX'Y$ into the final result. This is divided into three steps, which are described in the following three subsections.

4.2.1. *Superelement sorting*

We group elements into superelements (usually called blocks), of size $p = \lfloor n^{1/3} \rfloor$, that will be merged. If $|X'| \bmod p \neq 0$, i.e. there exist an undersized X-superelement, we place it at the left end of X'. If $|Y| \bmod p \neq 0$, we place the undersized Y-superelement at the right end of Y. Therefore, except for the first X-superelement of size $p_1 \leqslant p$ and the last Y-superelements of size $p_2 \leqslant p$, all X- and Y-superelements are of the same size p. The criterion for comparing two superelement are the keys of their last elements.

The task of this subsection is to sort the X- and Y-superelements stably (excluding the last (possibly) undersized Y-superelement) so that the keys of their last elements form a non-decreasing key sequence. We take Fig. 6 as an example to show the task. In Fig. 6, the italics c, d, f, g, a and f are the keys of the X-superelements 1–4 and the Y-superelements 1–2, respectively. Superelement sorting makes c, d, f, g, a and f form a non-decreasing key sequence, $a < c < d < f \leqslant f < g$, i.e. converts Fig. 6(a) into Fig. 6(b).

Usually, the first X-superelement is not full. To make it full, we view the last $p-p_1$ elements in the fifth C-block (denoted by C_4) as part of the first X-superelement. That is, the first X-superelement is viewed as the $p-p_1$ C_4-elements followed by the first p_1 X-elements. Thus the first superelement sequence is defined as all the X-superelements excluding the first $4p+p_1$ buffer elements, while the second superelement sequence is defined as all the Y-superelements excluding the last (possibly) undersized Y-superelement. Clearly, the two superelement sequences are already sorted. The sorting task here is actually to merge the two superelement sequences. Let each superelement corresponds to an element in Section 3. The merging routine shown in Section 3 is well suited to fulfill the task. Let $\tilde{n}$ denote the number of superelements to be merged. Then $\tilde{n} \approx n/p-6 = n^{2/3}-6$. Here we call a block in Section 3 a superblock, and divide the superelement sequences X and Y into superblocks of equal size $s = \lfloor n^{1/3} \rfloor \approx \tilde{n}^{1/2}$. Each superblock consists of s superelements, or $p \cdot s \approx n^{2/3}$ simple elements. The $2s$ X-superelements in the last two superblocks will be used in the same way as the last $2s$ X-elements in Section 2, i.e., for internal buffering. The buffer is still denoted by B.

The routine of Section 3 is unstable, but the task here requires to keep the stability of merging superelements. How do we solve the problem? Our solution is to keep track of the relative order of the X-superblocks and the relative order of the B-superelements. This can be achieved by swapping the C-elements, collected in Section 4.1, with the first simple elements in them. Prior to invoking the routine of Section 3, we exchange the ith C-element with the first simple element in the ith X-superblock, for $i = 1, 2, \ldots, t$, where t is the number of X-superblocks excluding two B-superblocks. Clearly $t < \tilde{n}/s \leqslant p$. Later the first C-block is called C_1. Similarly, we exchange the $(p+i)$th C-element with the first simple element in the ith B-superelement for $i = 1, \ldots, 2s$. These C-elements (i.e. the second and third C-block, since $s = p$) are denoted by C_2. We can select a current X-superblock (this corresponds to the selection of a current X-block in Section 3) in such a way: use C_1-elements, which are stored in the left ends of X-superblocks, to search the remaining X-superblocks for a superblock with the smallest C_1-element. This is feasible because C_1-elements in superblocks are distinct and initially sorted. Once an X-superblock becomes a current superblock, we put back its C_1-element and restore the content of the current superblock. In Section 3.1.11, we are required to select the smallest element b_1 from the remaining B-elements. Here we can select the smallest B-superelement in such a way: use C_2-elements, which are stored in the left ends of B-superelement, to search the remaining B-elements for a superelement with the smallest C_2-element. Once a B-superelement is placed in its final position, we put back its C_2-element and restore the content of the B-superelement.

To save the number of moves we use the floating hole technique, as described in Section 3.2. The creation of the first hole is the same as Section 3.2. That is, put the C-element (denoted by b) adjacent to the left of the first X-superelement aside, to create a hole. But data movement is different. For example, $o_c \leftarrow y_c \leftarrow b_c \leftarrow o_c + 1$ in Section 3.2 is viewed as hole+superelement $o_c \leftarrow$ superelement $y_c \leftarrow$ superelement $b_c \leftarrow$ superelement o_c, which can be implemented by the following operations: the $(i-1)$th $o_c \leftarrow$ the ith $y_c \leftarrow$ the ith $b_c \leftarrow$ the ith o_c, for $i = 1, \ldots, p$, where the ith o_c (y_c, b_c) refers to

the ith simple element in superelement o_c (y_c, b_c), and 0th o_c is a hole. The other data movements are similar. After merging, we move the last Y-superelement one position to the left to move the hole to the right end of Y. Then we place b, stored in a temporary memory, in the final hole.

It is easy to see that the size $4p$ of the buffer is sufficient to meet the requirement of the task. Since the number of X-superelements is less than $\tilde{n}$, the number of moves required for swapping and re-swapping the elements in the first position, in each superblock, and in each B-superelement is at most $O(\tilde{n}/s) + O(s)$. Further, it is trivial to conclude that we can stably merge the X- and Y-superelements in at most $3\tilde{n} \times p + O(\tilde{n}/s) + O(s) \leqslant 3n + O(n^{1/3})$ moves and $\tilde{n} + O((\tilde{n}/s)^2) + O(s^2) = O(n^{2/3})$ comparisons, using the merging routine of Section 3 in the above way.

4.2.2. *Local merging*

Below we view a superelement in the previous subsection as a block. To have the buffer elements in C_1 and C_2 not mixed up, we use the fourth C-block (denote by C_3) to do local merging. As shown in Fig. 6(b), suppose that the output of the previous subsection is of the form $C'Z_1 \ldots Z_i \ldots Z_t Y_L b$, where C' is the prefix of the buffer $C = C_1C_2C_3C_4, b$ is the C_4-element, each Z_i is either an X'-block or a Y-block, $|Z_i| = p$, Y_L is the last undersized Y-block, and $\text{last}(Z_i) \leqslant \text{last}(Z_{i+1})$, for $1 \leqslant i < t$. The goal of the subsection is to transform the $C'Z_1 \ldots Z_i \ldots Z_t Y_L b$ into $C_1C_2SC_3C_4$, where the C_1, C_2-elements are not mixed up, while the C_3, C_4-elements are mixed up, and S is the stable merging of X' and Y, which is implemented by merging locally $Z_1 \ldots Z_i \ldots Z_t Y_L$. This task can be done by a routine similar to the series-merging of [4]. Depending on whether the first X'-block is a full block or not, we adopt different strategies.

First we consider the case where the first X'-block is a full block. After superelement sorting, the buffer C, except for the last C-element b, is still placed in the left end. By scanning $Z_1 \ldots Z_t$ from left to right, we determine two series of elements to be merged. The first series consists of $Z_1, Z_2, \ldots, Z_k$, satisfying $Z_1 \lessdot Z_2 \lessdot Z_3 \lessdot \cdots \lessdot Z_{k-1} \lessdot Z_k$, where the notation $Z_j \lessdot Z_{j+1}$ (for $1 \leqslant j < k$) denotes $\text{Last}(Z_j) < \text{first}(Z_{j+1})$ if Z_j is a Y-block and Z_{j+1} is an X'-block (below it is showed how this question can be decided), and $\text{Last}(Z_j) \leqslant \text{first}(Z_{j+1})$ otherwise, and Z_k is the first block such that $Z_k \lessdot Z_{k+1}$ is false. The second series consists solely of Z_{k+1}. Now we merge these two series by repeating the following process: compare the leftmost unmerged element in the first series to the leftmost unmerged element in the second series, exchanging the smaller-keyed element with the left end of the elements of C_3C_4. We terminate this process when the last element of Z_k has been moved to its final position.

As before, we determine the next two series of elements to be merged. But the first block of the first series consists of the unmerged elements of Z_{k+1}. After the two series are located, we resume the above merge until the tail of the first series has been moved.

We repeat the above process of locating a series of elements and merging them until we can only locate one such series, which we merely move left so that the buffer C_3C_4 is moved to the right end.

Now we consider the case that the first X'-block is an undersized block. Assume that the first X'-block is Z_j. If $\text{last}(Z_{j-1}) < \text{first}(Z_j)$, we exchange $Z_1 \dots Z_{j-1}$ (Notice, $Z_1, \dots, Z_{j-2}$ and Z_{j-1} are Y-blocks) with C_3C_4' (where C_4' denotes the C_4-elements adjacent to the left of Z_1) by the routine of Section 2.4. Otherwise, we view $Z_1 \dots Z_{j-1}$ as the first series, and Z_j as the second series. As before, we merge these two series. The subsequent process is the same as the case above.

For the rightmost block, Y_L, the process is simple. If $\text{last}(Z_t) > \text{first}(Y_L)$, where Z_t is a block adjacent to the left of Y_L, we view it as the second series, which is merged with the first series located finally in the way described above. Whether $\text{last}(Z_t) > \text{first}(Y_L)$ or not, finally we exchange the unmerged elements in Z_{k+1} or $Z_{k+1}Y_L$ (where Z_{k+1} is the second series in the last series-merging) with the C_3, C_4-elements adjacent to their left so that the buffer C_3C_4 is moved to the right end.

In merging, we must know whether Z_i is an X'-block or a Y-block. This issue can be resolved by synchronizing this procedure and the block sorting of Section 4.2.1. That is, while sorting blocks, we locate two series. Once two series are formed, the routine of Section 4.2.1 passes the program control to this procedure. After local merging, we return the program control to the routine of Section 4.2.1, and then continue its subsequent process.

It is easy to apply the floating hole technique here. Before local merging, we create a hole by putting the first C_3-element aside. In merging, we always let the position adjacent to the right of the elements merged so far be a hole. This can be implemented by repeating such a process: place the smaller-keyed element in the hole, then place the C_3- or C_4-element adjacent to the right of the hole in the position occupied originally by the smaller-keyed element. At the end of local merging, we place back the first C_3-element in the last hole.

It is trivial to see that after using the floating hole technique this procedure consumes, at most, $2n$ moves and n comparisons. The binary-merge strategy of Hwang and Lin [5] shown in Section 2.6 can reduce the upper bound for the number of comparisons to $m_1(t+1) + \lfloor m_2/2^t \rfloor$, where $t = \lfloor \log_2(m_2/m_1) \rfloor$ and $m_1 \leqslant m_2$. In the series-merging, we can view the first series as a series of form WX^FY^F, or WY^FX^F, where W consists of X-blocks and/or Y-blocks, while X^F, Y^F consist of only X-blocks and Y-blocks, respectively. If it is WX^FY^F, it must be followed by an X-blocks, that is, the second series must be an X-blocks (denoted by X^S). Since WX^F precedes X^S, the merging of WX^FY^F and X^S can be reduced to the merging of Y^F and X^S. We fix $t = \lfloor \log_2(m_2/m_1) \rfloor$, view X^S, Y^F as the X and Y of Section 2.6, and keep the moving way here, to merge them by the routine of Section 2.6. This requires at most $n^x(t+1) + \lfloor n^y/2^t \rfloor$ comparisons, where n^x, n^y are the sizes of X^S and Y^F, respectively. Similarly, we can handle the case of WY^FX^F. Suppose that the number of calls to the routine of Section 2.6 is m, and at the ith call, the sizes of X and Y are n_i^x and n_i^y, respectively. Then the total number of comparisons is bounded by $\sum_{i=1}^{m}(n_i^x(t+1) + \lfloor n_i^y/2^t \rfloor) \leqslant m_1(t+1) + \lfloor m_2/2^t \rfloor$.

4.2.3. *Buffer insertion*

The goal of this subsection is to transform $C_1C_2SC_3C_4$, the output of the previous subsection, into the final result. From the above process, it is easy to see that the

elements of C_1C_2 have not been mixed up and are still sorted. Furthermore, the elements of C_1C_2 should precede the elements of C_3C_4. We therefore use the last element of C_2 for a binary search over S, to divide S into $S_LS_R : C_1, C_2$ are merged with S_L by a forward BLOCKMERGE, while C_3, C_4 are merged with S_R by a backward BLOCKMERGE. Whether forward BLOCKMERGE or backward BLOCKMERGE, we use leftmost insertion points and binary search. By Section 2.5, we get:

$O(\lambda \log n_L) \leqslant O(n^{1/3} \log n)$ comparisons for merging C_1C_2 with S_L,
$O(\lambda \log n_R) \leqslant O(n^{1/3} \log n)$ comparisons for merging C_3C_4 with S_R,
$n_L + 2\lambda^2 \leqslant n_L + O(n^{2/3})$ moves for merging C_1C_2 with S_L,
$n_R + 2\lambda^2 \leqslant n_R + O(n^{2/3})$ moves for merging C_3C_4 with S_R,

where n_L, n_R denote the sizes of S_L and S_R, respectively, and $n_L + n_R \leqslant n$.

Before merging C_3, C_4 with S_R, we require to sort C_3C_4 in-place. Using a heap sort, this requires at most $O(\lambda \log \lambda) \leqslant O(n^{1/3} \log n)$ comparisons and moves. To sum up, we get at most $O(n^{1/3} \log n)$ comparisons and $n + O(n^{2/3})$ moves for embedding C into S.

If X is exactly C, the processes in the previous two subsection are not required. We merge directly X (i.e. C) with Y by a forward BLOCKMERGE. By an analysis similar to above, this takes at most $O(n^{1/3} \log n)$ comparisons and $n + O(n^{2/3})$ moves.

4.3. Merging in the presence of less than λ distinct keys

Assume that β, the number of distinct elements in X is less than $\lambda = 5\lfloor n^{1/3} \rfloor$, and X is transformed into CX', where C is a buffer of size β. The goal of this section is to transform $CX'Y$ in the case into the final result. Since $\beta < \lambda$, the previous method is not suited for this case. However, this resembles a special case described in the stable merging routine of Geffert et al. (see Section 4.4 of [2]). Hence, we utilize a technique similar to the stable merging of Geffert et al. to perform the merge of X' and Y. The difference between our algorithm and the algorithm of Geffert et al. is that our block sorting employs selection sort. This is possible because the number of blocks to be sorted here is $O(n^{1/3})$, which is much fewer than the number of blocks in [2]. The benefit of applying selection sort is to make our algorithm simpler than the algorithm of Geffert et al.

4.3.1. Block sorting

We divide X' and Y into blocks of equal size $q = \lceil m_1/\beta \rceil$. More precisely, we view X' and Y as sequences with the form $X' = X_1 \ldots X_i \ldots X_u$ and $Y = Y_1 \ldots Y_j \ldots Y_v$, where $|X_i| = |Y_j| = q$ for $2 \leqslant i \leqslant u$ and $1 \leqslant j < v$, $|X_1| \leqslant q$, $|Y_v| \leqslant q$. Clearly $u = \lceil |X'|/q \rceil \leqslant \beta < \lambda$. We can image X_1 as a block containing, at the left end, the fictitious element $(-\infty)$ smaller than any other element, and Y_v as a block containing, at the right end, the fictitious element $(+\infty)$ larger than any other element. Unlike Section 4.2.1, we define the key of each X-block as the key of its *first* element, and the key of each Y-block as the key of its *last* element. The task of the subsection is to sort stably the sequence of blocks $X_1 \ldots X_u Y_1 \ldots Y_v$ so that the keys of the blocks form a non-decreasing key sequence. To be more precise, the task is to transform $C'bX_1 \ldots X_i \ldots X_u Y_1 \ldots Y_j \ldots Y_v$ (see Fig. 7(a)) into $C'X_1SY_vb$ (see Fig. 7(b)), where b is the last buffer element, $C = C'b$, and S is the stable block sorting of the sequence $X_2 \ldots X_u Y_1 \ldots Y_{v-1}$. By

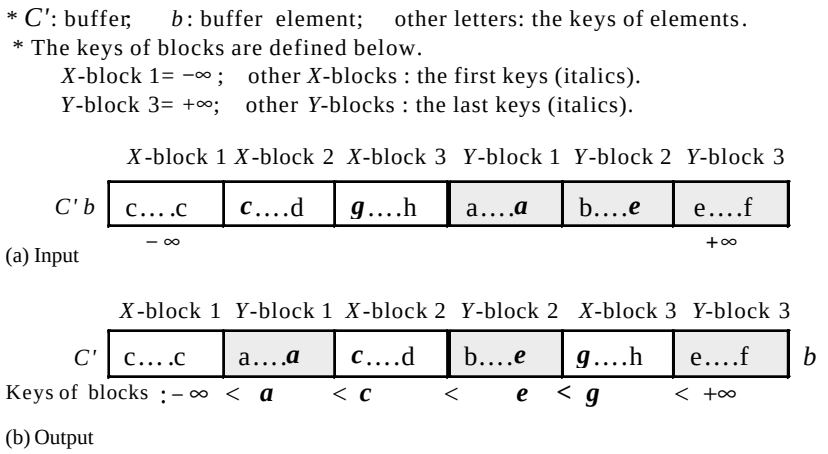


Fig. 7. The block sorting of Section 4.3.1.

the assumption, $\mathrm{Key}(X_1)=-\infty$ and $\mathrm{Key}(Y_v)=+\infty$. Hence, the main work of the subsection is to sort stably q-sized blocks $X''=X_2\ldots X_u$ and $Y'=Y_1\ldots Y_{v-1}$. As shown in Fig. 7, if $\mathrm{Key}(X_2)=c$, $\mathrm{Key}(X_3)=g$, $\mathrm{Key}(Y_1)=a$, $\mathrm{Key}(Y_2)=e$, the block sorting converts $C'bX_1X_2X_3Y_1Y_2Y_3$ into $C'X_1Y_1X_2Y_2X_3Y_3b$.

We shall utilize selection sort containing the floating hole technique to perform the block sorting. We keep the algorithm stable by keeping track of the original order of the X''-blocks. This is easily achieved by exchanging the last elements in X''-blocks with the corresponding C-elements. To be able to apply the floating hole technique, we initially create a hole e by putting b aside. Now we proceed to merge X'' with Y' by selection sort as follows. We find the block X_s with the smallest C-element, stored in the right end, by scanning the unmerged X''-blocks from left to right. Let Y_s denote the first block of the unmerged Y'-blocks, Z denote the block immediately to the right of the hole e, and $\mathbf{min}(X_s,Y_s)$ denote X_s if $\mathrm{first}(X_s)\leqslant\mathrm{last}(Y_s)$ and Y_s otherwise. We swap $\mathbf{min}(X_s,Y_s)$ and eZ by the floating hole technique of Section 2.3, using $2q$ moves. If $\mathbf{min}(X_s,Y_s)=Z$, this operation uses q moves. The hole is now in the rear of the merged result (i.e. in the right of Z). If $\mathbf{min}(X_s,Y_s)=X_s$, we reswap the X_s-element in C with the C-element in X_s, to restore their original contents. This process is repeated until the last X''-block, X_u, has been placed in the final position. Finally we sequentially move all the remaining Y'-elements including Y_v one location to the left. The hole is now in the rear of Y_v. We halt this process by placing the b in the hole.

Clearly, this phase needs at most $2n+\mathrm{O}(\beta)=2n+\mathrm{O}(n^{1/3})$ moves and $\mathrm{O}(u^2+v)$ comparisons. But if we determine the insertion points by binary search on Y', the number of comparisons can be bounded by $\mathrm{O}(u^2+u\log v)=\mathrm{O}(n^{2/3})$.

4.3.2. *Local merging*

It is easy to see that the previous subsection has transformed $X'Y$ into a sorted block sequence $X_1Y_1X_2Y_2\ldots X_kY_k$, where each X_i is a series of some consecutive X'-blocks, each Y_i a series of some consecutive Y-blocks. Let $\mathrm{FirstB}(X)$ denote the first block of

X, LastB(X) the last block of X, $K_1(X)$ the first key of X, $K_L(X)$ the last key of X. Since $X_1Y_1X_2Y_2\ldots X_kY_k$ is the output of the previous subsection, we have

$$\begin{aligned} K_L(\text{LastB}(Y_{i-1})) &< K_1(\text{FirstB}(X_i)) \leqslant K_1(\text{LastB}(X_i)) \\ &\leqslant K_L(\text{FirstB}(Y_i)) \leqslant K_L(\text{LastB}(Y_i)) \end{aligned} \tag{1}$$

for $i=1,2,\ldots,k$ (take $K_L(\text{LastB}(Y_0))=-\infty$). The task of the subsection is to transform the $X_1Y_1X_2Y_2\ldots X_kY_k$, into the stable merging of X' and Y. This can be achieved by the same local merging as in Section 4.6 of [2]. This routine may be described as follows.

Let F_i denote the first block of Y_i, and G_i the remainder of Y_i, i.e. $Y_i=F_iG_i$. For each i, we divide X_i into three parts, namely, $X_i=C_iD_iB_i$ such that $\text{last}(F_i)<\text{first}(D_i)$ and $\text{last}(D_i)\leqslant\text{last}(G_i)<\text{first}(B_i)$. By (1), it is easy to conclude that the size of D_iB_i is less than that of a block. i.e., $|D_iB_i|<q=\lceil m_1/\beta\rceil$. Thus, we can determine the boundaries of D_i by binary search over the last block of X_i , using $O(\log q)$ comparisons. The merge of X' and Y can be performed by the following pseudo-code.

```
for i = 1 to k do
   merge D_iB_i with F_iG_i i.e. convert C_iD_iB_iF_iG_i to C_iF_iH_iB_i
   merge B_{i-1}C_i (assume B_0 is empty) with F_i
end for
```

where H_i is the merge of D_i and G_i. To merge D_iB_i with F_iG_i (i.e. Y_i), we invoke a forward BLOCKMERGE, using leftmost insertion points and binary search. Suppose that D_i has d_i distinct elements. Then the number of block exchanges required by merging D_iB_i with Y_i is at most d_i+1, where the last exchange is used for moving B_i to the right end. By Section 2.5, this requires $O(d_i\log|Y_i|)$ comparisons and at most $|Y_i|+2(d_i+1)|D_iB_i|<|Y_i|+2(d_i+1)q$ moves. To merge $B_{i-1}C_i$ with F_i, we invoke a backward BLOCKMERGE, using rightmost insertion points and binary search. This requires $O(t_i\log|B_{i-1}C_i|)$ comparisons and at most $|B_{i-1}C_i|+2t_i|F_i|=|B_{i-1}C_i|+2t_iq$ moves, where t_i is the number of distinct elements in $B_{i-1}C_i$.

It is easy to see that $\sum_{i=1}^{k}|B_{i-1}C_i|<|X|$, and $\sum_{i=1}^{k}(t_i+d_i)\leqslant\beta$. Recall that X' consists of only $u\leqslant\beta$ blocks, hence we have $k\leqslant u<\beta$. Thus, we can obtain that the total number of comparisons required by the local merging is bounded by $O(\beta\log n)\leqslant O(\lambda\log n)=O(n^{1/3}\log n)$, and the total number of moves is bounded by $\sum_{i=1}^{k}(|Y_i|+2(d_i+1)q+|B_{i-1}C_i|+2t_iq)<|Y|+|X|+2q\beta+2qk\leqslant m_2+5m_1$, using $q=\lceil m_1/\beta\rceil$.

During the local merging, we must know the boundaries of each X_i and Y_i. This problem can be solved by synchronizing this procedure and the block sorting of Section 4.3.1. That is, once X_iY_i is formed, the routine of Section 4.3.1 passes the program control to this procedure. After local merging, we return the program control to the routine of Section 4.3.1, and then continue its process. Of course, in switching the program control, we have to store necessary information such as the boundaries of X_i and Y_i, and the position of the hole etc. To store such information, a constant memory is sufficient.

4.3.3. Buffer insertion

Let S denote the stable merging of X' and Y. The output of the previous subsection is of the form $C'Sb$, i.e. C', the first $\beta-1$ C-elements, is located in front of the merged result, and b, the last C-element (also the largest C-element), is in the rear of the merged result. Recall that C' is sorted. The goal of this subsection is to transform $C'Sb$ into the final result. Now we merge C' with S by a forward BLOCKMERGE using leftmost insertion points and binary search. By Section 2.5, this merging requires at most $O(\beta \log n) \leqslant O(n^{1/3} \log n)$ comparisons and $l+O(\beta^2) \leqslant l+O(n^{2/3})$ moves, where l is the number of the elements smaller than the last C'-element. We determine the final location of b by a binary search over the last $n-l-1$ element of the output result, and move sequentially all elements no smaller than b one location to the right and then place b in its final position. Clearly, the overall number of comparisons required by the procedure is bound by $O(n^{1/3} \log n)$, and the overall number of moves is bounded by $n + O(n^{2/3})$.

4.4. Time complexity

Now we derive the time complexity of our stable merging algorithm from the previous sections. By Section 4.1, the extraction of buffer C takes at most $O(n^{1/3} \log n)$ comparisons, and $m_1 + O(n^{2/3})$ moves. Let β be the number of distinct X-elements. When $\beta < \lambda = 5\lfloor n^{1/3} \rfloor$, we employ the routine of Section 4.3 to stably merge CX' with Y into one ordered sequence. By Sections 4.3.1–4.3.3, the number of comparisons required in merging CX' with Y is bounded by $O(n^{2/3})+O(n^{1/3} \log n)+O(n^{1/3} \log n) = O(n^{2/3})$, and the number of moves is bounded by $(2n+O(n^{1/3}))+(m_2+5m_1)+n+O(n^{2/3}) = 8m_1 + 4m_2+o(n)$, which is less than or equal to $6n+o(n)$ when $m_1 \leqslant m_2$. When $\beta \geqslant \lambda$, we employ the routine of Section 4.2 to stably merge CX' with Y into one ordered sequence. By Sections 4.2.1–4.2.3, the number of comparisons required in merging CX' with Y is bounded by $O(n^{2/3}) + (m_1(t+1) + m_2/2^t) + O(n^{1/3} \log n) = m_1(t+1) + m_2/2^t + o(n)$, where $t = \lfloor \log_2(m_2/m_1) \rfloor$, and the number of moves is bounded by $(3n + O(n^{1/3})) + 2n + (n + O(n^{2/3})) = 6n + o(n)$.

To summarize, our algorithm requires at most $m_1(t+1) + m_2/2^t + o(n)$ comparisons and $6m_2 + 7m_1 + o(n)$ moves when $m_1 \leqslant m_2$.

5. Conclusions

In this paper a better algorithm was devised by crossing Huang and Langston's algorithm with the algorithm of Geffert et al. Up to now, the known stable in-place merging with the best upper bound for the time complexity is the algorithm of Geffert et al., which merges two ordered sequences in at most $m_1(t+1) + m_2/2^t + o(m_1)$ comparisons ($t = \lfloor \log_2(m_2/m_1) \rfloor$) and $5m_2 + 12m_1 + o(m_1)$ moves, where m_1 and m_2 are the sizes of two ordered sublists to be merged, and $m_1 \leqslant m_2$. Our algorithm can do this at most $m_1(t+1) + m_2/2^t + o(n)$ comparisons and $6m_2 + 7m_1 + o(n)$ moves. One can easily see that our algorithm is more efficient than that of Geffert et al., whenever $m_2 \geqslant m_1 > m_2/5$, since then $6m_2 + 7m_1 < 5m_2 + 12m_1$. If $m_1 \geqslant m_2$, we use a

symmetrical algorithm, viewing the rightmost element as the beginning of the array. Here the analogous condition is $m_2 > m_1/5$. As is well known, in merge sorting, two sequences to be merged are more frequently almost equal. Hence, our algorithm can achieve, more efficiently, the merge sorting. Another significant characteristic is that our algorithm is simpler, which makes it easier that one develops a practical stable in-place merging.

So far, unstable merging with minimum data movement requires $3n + o(n)$ moves (see [2]). Namely, in terms of the number of moves, our stable merging exceeds it by a factor of about two. This is the best among the known upper bounds. However, it is unclear whether our algorithm reaches the lower bound for the number of moves. This is left as an open problem.

Acknowledgements

The author is grateful to the referees for their careful reading and useful suggestions that helped to improve the contents and presentation of the paper, especially for suggesting the modification in Section 4.2 for the superelement merging.

References

[1] K. Dudzinski, A. Dydek, On a stable minimum storage merging algorithm, Inform. Process. Lett. 12 (1981) 5–8.
[2] V. Geffert, J. Katajainen, T. Pasanen, Asymptotically efficient in-place merging, Theoret. Comput. Sci. 237 (2000) 159–181.
[3] B.-C. Huang, A. Langston, Practical in-place merging, Comm. ACM 31 (1988) 348–352.
[4] B.-C. Huang, M.A. Langston, Fast stable merging and sorting in constant extra space, Comput. J. 35 (1992) 643–650.
[5] F.K. Hwang, S. Lin, A simple algorithm for merging two disjoint linearly ordered sets, SIAM J. Comput. 1 (1972) 31–39.
[6] D.E. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley, Reading, MA, 1973.
[7] M.A. Kronrod, An optimal ordering algorithm without operational field, Soviet Math. Dokl. 10 (1969) 744–746.
[8] H. Mannila, E. Ukkonen, Simple linear-time algorithm for in situ merging, Inform. Process. Lett. 18 (1984) 203–208.
[9] J.S. Salowe, W.L. Steiger, Simplified stable merging tasks, J. Algorithms 8 (1987) 557–571.
[10] A. Symvonis, Optimal stable merging, Comput. J. 38 (1995) 681–690.
[11] L. Trabb Pardo, Stable sorting and merging with optimal space and time bounds, SIAM J. Comput. 6 (1977) 351–372.